The FTIO Benchmark

Frederick C. Fagerstrom Christopher L. Kuszmaul

December 27, 1999

Contents

Abstract

We introduce a new benchmark for measuring the performance of parallel input/ouput. This benchmark has flexible initialization, size, and scaling proporties that allows it to satisfy seven criteria for practical parallel I/O benchmarks.

We obtained performance results while running on the a SGI Origin2000 computer with various numbers of processors: with 4 processors, the performance was 68.9 Mflop/s[1] with .52 of the time spent on I/O, with 8 processors the performance was 139.3 Mflop/s with .50 of the time spent on I/O, with 16 processors the performance was 173.6 Mflop/s with .43 of the time spent on I/O, and with 32 processors the performance was 259.1 Mflop/s with .47 of the time spent on I/O.

1 Introduction

The NAS[2] Parallel Benchmarks [1] have set the standard for measuring performance of distributed computer systems. These benchmarks do not measure I/O. A small extension of the NAS Benchmarks, called NHT-1 I/O [2], has provided a valuable, if limited window into the performance of I/O on parallel systems. We have further extended the NAS Benchmarks to include a full out-of-core implementation of the FT Benchmark.

In [3], six criteria are presented as characterizing an ideal I/O Benchmark. For parallel computing, the absolute I/O performance matters, but relative I/O performance is often as important. A subset of the criteria in [3] and two additional criteria are given for a practical I/O Benchmark of relative I/O performance on parallel systems:

- The benchmark should provide information about any performance weakness in the system.
- The benchmark should scale well both in terms of the number of processors and the size of the problem.
- The benchmark should not favor one particular system architecture over another.
- The benchmark should maintain a tight specification.

Two criteria from [3] that are not suited for a parallel I/O Benchmark are that the benchmark should be I/O limited, and that the benchmark should be useful for predicting performance for a wide range of applications.

The benchmark should stress I/O, but not be I/O limited. I/O is rarely done alone, without interspersed computations or parallel communication, and therefore a benchmark that strives to return relevant performance data should use I/O similarly to how it is actually used by applications.

Although it would be desirable for an I/O benchmark to be useful for predicting performance of a wide variety of applications, systems are generally used for a specific application and optimal I/O performance I/O performance for that application matters most. A more focused benchmark will give more relevant results.

In addition, we value following criteria: The benchmark should not be independent of the balance between computation, interprocessor communication and I/O speeds. Rather than having the benchmark I/O-bound, it should give an upper bound of the performance of out-of-core applications. The benchmark should report relative performance—the system balance is almost as important as the absolute performance, since balance speaks to scalability.

2 Design

In practice, a good I/O benchmark indicates that one I/O system is better than another for a specific type of computation. The FTIO benchmark gauges the relative performance of I/O to non-I/O operations while performing a 2-dimensional Discrete Fourier Transform (DFT). The FTIO stores most data out-of-core and reads in and operates on only small segments of the data at once.

[1] Mflop/s is million floating point operations per second.

[2] NAS is Numerical Aerospace Simulation Facility.

A 2-dimensional DFT is defined as follows where $\tilde{x}(m,n)$ specifies the element at position (m,n) in the MxN input matrix and $\tilde{X}(k,l)$ specifies the element at position (k,l) of the MxN output matrix:

$$\tilde{X}(k,l) = \sum_{m=1}^{M-1} \sum_{n=0}^{N-1} \tilde{x}(m,n) W_M^{-km} W_N^{-ln} \tag{1}$$

with

$$W_M = e^{-\sqrt{-1}(2\pi/M)}$$
$$W_N = e^{-\sqrt{-1}(2\pi/N)}$$

Thus, to perform a 2-dimensional DFT, it is sufficient to perform two sets of 1-dimensional DFTs, one on the input matrix's rows and then one on the matrix's columns or vice versa.

Basing the FTIO on the DFT is reasonable. The FTIO Benchmark's goal is to gauge the I/O performance during physical modeling and simulation calculations that involve data that must be stored out-of-core. Large numerical calculation problems rely on reading in small portions of data at once. The reads may be done sequentially, but are often not, and therefore factoring in the cost of alternating between seek operations and read/write operations is important. The 2-dimensional DFT is well-suited for an implementation that involves both seeking and sequential reading/writing of coherent chunks of data. Furthermore, the 1-dimensional phase involves interprocessor communication and calculation as would the computation phase of a more general class of algorithms that would include many used for physical modeling or computational fluid dynamics. Although the topologies of communication or the pattern of file access in any specific computation may differ from that of the DFT, the similarity in algorithms makes the results relevant.

One implementation for the 2-dimensional DFT is to perform a series of 1-dimensional DFTs—one to each row of the matrix, transpose the matrix, and perform another series 1-dimensional DFTs to the rows of the matrix. Given that $\tilde{x}(m,n)$ specifies the input matrix, the output matrix is specified by $\tilde{X}(l,k)$, that is, the transpose of the output matrix given in (1).

3 Implementation

3.1 Overview

The FTIO Benchmark performs a 2-dimensional Fourier transform on a large set of data partitioned among the various processors used for computation. This section includes mention of the Tools (3.2) used to implement the FTIO, explanation of the FTIO's input parameter (3.3) and discussion of the implementation for each section of the FTIO computation: Initialization (3.4), DFT Series (3.5), Transposition (3.6) and Verification (3.7).

3.2 Tools

The FTIO Benchmark was written in C using the Message Passing Interface (MPI) and the Fastest Fourier Transform in the West (FFTW). The program **perfex** was used to collect cycle and floating point operation (flop) count data for the FTIO Benchmark, and the Portable Batch System (PBS) was used to submit FTIO to the SGI Origin2000 Cluster.

3.2.1 MPI

MPI is a library specification standard for message-passing on massively parallel machines and on workstation clusters. Using MPI, data is explicitly distributed among the processors. The MPI standard and related documents are available [5]. FTIO uses MPI to transfer data among processors and to time computation, communication and I/O.

Data-transfer for the Transposition phase is done using `MPI_Sendrecv_replace()` which is used to implement simultaneous data-exchanges between processors. During the verification phase, right before FTIO exits. `MPI_Send()` and `MPI_Recv()` are used to transfer verification and output data to processor 0. FFTW DFT during the DFT Series is the only other place where interprocessor communication is used.

Timing is done based on wall time, not processor time. The function `MPI_Wtime()` is used before and after the region of code that is going to be timed. Because interprocessor communication and I/O reads and writes[3] are blocking, the timing data is accurate. The runtime of the FTIO program differs from the runtime returned as part of the benchmark because the benchmark does not include overhead for program startup and shutdown for running FTIO which includes `MPI_Init()` and `MPI_Finalize()`.

3.2.2 FFTW

FFTW is a C subroutine library for computing the DFT of complex or real vectors. FTIO uses the FFTW routines for computing the DFT of a complex, 1-dimensional series of data distributed among the processors using MPI. Good FFTW documentation is available [4].

FFTW uses a data-structure called a plan to perform a DFT. The plan is constructed to perform a forward DFT which means that $W_N = e^{-\sqrt{-1}(2\pi/N)}$, rather than $W_N = e^{\sqrt{-1}(2\pi/N)}$, is used in the transform: $Y(k) = \sum_{n=1}^{N-1} X(n) W_N^{nk}$. Estimation rather than measurement is used to determine the fastest plan for performing the DFT. Since only two DFTs are performed, the time to perform the measurement is greater than the increase in performance it would yield. Also, using measurement may result in a plan that distributed the data among the processors in a manor incompatible with the transpose algorithm.

3.2.3 `perfex`

The utility `perfex` can be used to obtain information on hardware counters. During the development and testing of FTIO, we used `perfex` to obtain the number of cycles and flops used by the executing code. To insure the flop count returned is right and does not undercount due to the presence of a MADD[4] instruction on SGI Origin2000 architecture, FTIO is compiled with the option `-TARG:madd=OFF`. Also, using `perfex` required a flag in the PBS script.

The `perfex` command is executed as follows to count both cycles and flops:

```
perfex -mp -e 21
```

3.2.4 PBS

PBS is a flexible batch queuing system [6]. An example script for queuing a job is given below:

```
#PBS -l ncpus=64
#PBS -l walltime=1200
#PBS -l hpm=1
cd $TMPDIR
mpirun -np 64 perfex -mp -e 21 /u/fredf/finftio/ftio 8
```

The actual number of processors used (`ncpus=64`) and the number of processors perceived by MPI (`-np 64`) are both 64; the perceived and actual number should be the same for FTIO benchmarking. The `walltime` is in seconds. The `hpm=1` is required by `perfex`. The `$TMPDIR` should be the location of the disk where the processors temporary files will be kept. The FTIO Benchmark will test I/O performance of the I/O-space specified by `$TMPDIR`. The command `/u/fredf/finftio/ftio 8` executes the benchmark and passes it the input parameter 8.

3.3 Input Parameter

The input parameter determines the width/height of the square matrix used by the FTIO Benchmark. The input parameter is passed in as a command-line argument, and determines the width/height of the matrix N. Let i be the input parameter, and let c be the number of processors used (perceived by MPI). Then

[3] Output is flushed within the write timing-blocks.

[4] MADD is a combined multiply and add in one instruction.

$N = i^4c^2$. Because the FFTW plan must specify that data be distributed evenly among the processors, i and c must satisfty the two assertions: $c|i^2$ and $i^2 \geq c$. If either assertion fails, the FTIO Benchmark will not run. If i^2 is too small or if i^2 is not divisible, then FFTW does not consistently distribute the date correctly.

3.4 Initialization

During the initialization phase, each processor calculates and saves to disk its portion of the input matrix. With the matrix size N specified by the input parameter, the function $\tilde{x}(i,j)$ defines the values of the NxN input matrix where i is the column and j is the row:

$$\tilde{x}(i,j) = e^{\sqrt{-1}(2\pi i)/N} + e^{\sqrt{-1}(2\pi j)/N} + imp(i,j) + spike(i,j) \tag{2}$$

with $1 \leq i \leq N$, $1 \leq j \leq N$, and

$$imp(i,j) = \begin{cases} 1 & \text{if } j = (i+1) \bmod N \\ 0 & \text{otherwise} \end{cases}$$

$$spike(i,j) = \begin{cases} 1 & \text{if } (i=0) \text{ AND } (j=1) \\ 0 & \text{otherwise} \end{cases}$$

The given function does not become sparse during computation; therefore a system that can take advantage of simple compression of all-zero data arrays in I/O is not favored. There is also a quickly-calculatable function that returns the values of the output matrix which is used for Verification.

3.5 DFT Series

A DFT series is used after initialization to calculate the DFT of each row of the matrix, and then again after transposition to calculate the output matrix.

During the DFT series, each row's DFT is calculated: Each processor reads in its section of the row and uses an FFTW routine to calculate the DFT of the row. The DFT calculation is in-place; thus after the DFT each processor contains its portion of the result in-core. The processor then saves the result to disk.

3.6 Transposition

The transpose is done by dividing the matrix into Squares. The elements in each Square are transposed, and then the Squares themselves are transposed. The algorithm is based on an extension of the implication (3).

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \Longrightarrow M^T = \begin{pmatrix} A^T & C^T \\ B^T & D^T \end{pmatrix} \tag{3}$$

where A, B, C, and D are rectangular sub-regions (matrices) that tile[5] M

Transposing the elements in a Square is done without interprocessor communication because each Square resides on a single processor, and is carried out by compartmentalizing the elements into Mini-Squares, as illustrated in Figure 1. Each Mini-Square is read into memory, transposed and then written out to the proper transposed location; transposing the elements of a square is also based on the implication (3). A temporary buffer is used to read in the Mini-Square that has not yet been transposed, while its transposed conjugate[6] overwrites it. Any Mini-Square that is its own conjugate lies along the Main Diagonal and is written to the same location from which it was read.

Once the elements in each Square have been transposed, entire Squares are exchanged. Every processor must specify which of its Squares to exchange and with which processor to exchange; thus a scheme based on Diagonals is used. (Figure 1 contains an example of the use of Diagonals.) Each processor iterates throughout the Diagonals, and a given Diagonal specifies which Square will be exchanged. If the Square to exchange is number i, then the processor with which to exchange is processor number i. MPI is used to exchange data.

[5] A set of sub-regions tile a matrix M if every element of M is in one and only one sub-region.

[6] The conjugate (Mini-)Square of a (Mini-)Square at location (x,y) is the (Mini-)Square located at (y,x).

3.7 Verification

During the verification phase, each processor uses the verification function (4) to calculate the difference between the output value in the matrix and the expected value. The total difference is reported in the output and can be used to gauge the accuracy of the system's calculations. The function *expected*(i,j) defines an NxN matrix where i specifies the column and j specifies the row.

$$\mathit{expected}(i,j) = e^{-\sqrt{-1}(2\pi i)/N} + \mathit{diag}(i,j) + \mathit{spikes}(i,j) \tag{4}$$

with $1 \leq i \leq N$, $1 \leq j \leq N$, and

$$\mathit{diag}(i,j) = \begin{cases} e^{-\sqrt{-1}(2\pi j)/N} & \text{if } (i+j) \bmod N = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathit{spikes}(i,j) = \begin{cases} 1 & \text{if } (i=1) \text{ AND } (j=0) \\ 1 & \text{if } (i=0) \text{ AND } (j=1) \\ 0 & \text{otherwise} \end{cases}$$

4 FTIO Benchmark Meets Criteria

The I/O-time ratio returned will indicate how much I/O is slowing down the computation. For example, if the I/O Fraction is significantly above .5, then an emphasis should be placed on increasing I/O performance.

FTIO is designed not to take advantage of the performance on any specific system. FFTW estimate is used rather than measure to provide a moderate level of system-tailored performance enhancement. The rest of the code, however, is written to make it difficult for one system to do unusually well due to some narrow optimization.

FTIO scales well in terms of the number of processors and the size of the problem. Because FTIO results are consistent throughout a range of matrix sizes, it is not necessary to run extremely large-scale runs, as a sufficiently large run is much smaller. Also, the FTIO may be run on an arbitrary number (within the constraints of MPI) of processors greater than 1. The matrix size can also be made smaller or larger by varying the input parameter.

The FTIO Benchmark is fixed and specified. Specification of running environments and system characteristics must be included whenever reporting results of the FTIO Benchmark.

The FTIO returns the I/O Fraction which is a measure of I/O's relative performance.

5 FTIO Performance on SGI Origin2000

FTIO was run using 4, 8, 16, and 32 processors on an SGI Origin2000 Cluster with 64 195 MHZ IP27 processors each having 250 MB of memory. The disks available for I/O were two 100 GB scratch-space disk arrays. Each disk array was set up to be equally accessible by any processor in the cluster. To determine whether it would be possible to obtain representative data by running FTIO on relatively small matrices, we ran FTIO using a range of input parameters to observe how the I/O Fraction (5.1) and Cycles per flop (5.2) change with the matrix size. We estimated the optimal matrix size based on results from 5.1 and 5.2, and assembled benchmark results for the various numbers of processors (5.4). We also developed a benchmark size-specifying function, assembled results using it and compared them to the optimal results (5.5).

5.1 I/O Fraction

Figure 2 shows the I/O Fraction obtained from running the FTIO on 4, 8, 16 or 32 processors with varying input matrix sizes.

The data points graphed represent the averages of anywhere from 1 to 4 runs, and therefore anomalous I/O-performance will not affect them as much as if they were all data points from single runs.

For all the matrix sizes the I/O Fractions obtained from the runs stay within certain bands that are not more than .23 in width. The data points obtained while running on 4 processors stay within a band from

.44 to .67. With 8 processors, the data points stay within a band from .42 to .59. The 16-processor data stay within a band from .37 to .57. And the 32-processor data are in a band from .37 to .47.

5.2 Cycles per flop

A pattern emerges in the graphs of cycles per flop in Figure 3. Averages of the input data are used as in the previous section.

The number of cycles per flop gives an estimation of how many cycles were spent on I/O or interprocessor communication as opposed to floating point computations. The graphs suggest asymptotic convergence. The smaller matrix sizes have less FTIO related computation to do and are therefore more affected by transient I/O conditions and program startup/shutdown overhead.

Excluding the small matrix sizes the data stay within bands no wider than 121. With 4 processors, the all data points except the first are between 16 and 55. With 8 processors, excluding the data point, data all fall between 19 and 46. With 16 processors, excluding the first data point the band is much larger, spanning from 22 to 143. And with 32 processors, excluding the first data point, the band is from 38 to 62.

Because there are bands that the data stay within and the graphs indicate asymptotic convergence, it may be possible to find the I/O Fraction that the FTIO would converge to by running the FTIO using a smaller input parameter.

5.3 Determining the Input Parameter

5.3.1 Hypothesis

The hypothesis on which the FTIO is based is that there is some I/O Fraction that characterizes a system's relative I/O performance Since the data presented using various input sizes have I/O Fractions that vary, it may be presumed that the hypothesis does not hold perfectly. Yet there is some convergence and regularity to the I/O Fraction data and cycles per flop data obtained. I/O Fraction bands (5.1) suggest that some representative I/O Fraction exists.

5.3.2 Method

The goal is to select a data point that is close to the middle of the band to find a representative I/O Fraction. Data points with larger matrix sizes should be preferred over data points with smaller matrix sizes if their I/O Fraction values differ significantly. However, if they are the same, the smaller matrix size will allow the I/O Benchmark to run in less time.

5.3.3 Secondary Goal

FFTW DFT is based on code that runs faster or slower depending on the prime factor decomposition of the input parameters. Thus FFTW DFT introduces another level of variation into the I/O Fraction data obtained. Therefore, as a secondary goal the matrix sizes one chooses should result in input parameters that have similar prime factor decompositions.

5.3.4 Results

We select the third data point of the 32-processor run data because it is the largest input value available. The third data point for 32 processors corresponds to an input parameter of 24, which is an input parameter that is associated with a valid input for each other number of processors. And thus, to satisfy the secondary goal of having similarly prime factors for each input parameter to make performance comparison between the different numbers of processors significant, we use 24 as the input parameter for every number of processors.

5.4 Benchmark Results

A more accurate I/O Fraction can be obtained by running the benchmark multiple times and then statistically analyzing the results. Using the input parameter 24 determined in the previous section, we measured the I/O

Fraction multiple times for each number of processors. The average of the I/O Fractions is the representative I/O Fraction for the number of processors. Table 1 contains the results.

Number of Processors	4	8	16	32
Estimated Input Parameter	24	24	24	24
Average of I/O Fractions	.52	.50	.43	.47
Total Mflop/s	68.9	139.3	173.6	259.1

Table 1: I/O Fraction Benchmark Results

Graphs of the data collected for 4 processors, 8 processors, 16 processors and 32 processors are in Figure 4.

5.5 Specified Matrix-Size Benchmark Results

Graphing the I/O Fractions for various input parameters, then picking an input parameter (and thus matrix size) large enough to yield accurate enough results takes time and relies on human judgment. It would be better to have a way to pick the input parameter to directly calculate the I/O Fraction. We hypothesize that the following function will give input parameters that yield good I/O-Fraction results:

$$inputParameter(c) = \text{the lowest valid input parameter greater than or equal to } \sqrt{c} + 1 \qquad (5)$$

where c is the number of processors.

We collected I/O Fraction data using the input-parameter specified by (5) on 4 processors, 8 processors, 16 processors, and 32 processors. Graphs of the data obtained are in Figure 5 and the I/O Fraction data is in Table 2.

Number of Processors	4	8	16	32
Specified Input Parameter	4	4	8	8
Average of I/O Fractions	.58	.51	.56	.39
Difference	.06	.01	.13	−.08
Error	11.5%	2.0%	30.2%	17.0%

Table 2: I/O Fraction Specified Matrix-Size Results. Difference is the difference between the I/O Fraction here and the corresponding I/O Fraction in Table 1. Error is the Difference divided by the actual I/O Fraction from Table 1.

6 Conclusion

We have introduced criteria for a benchmark that returns I/O performance relative to the other operations required for parallel computation such as interprocessor communication and floating point calculation. We discuss the design and implementation of the FTIO Benchmark and conclude that it meets the criteria for measuring relative I/O performance data. We test the performance of a SGI Origin2000 Cluster, and in the process develop a function that specifies an input parameter that yields an I/O Fraction within 30% of the actual benchmark value.

We look forward to having others try the FTIO Benchmark on a variety of systems. To obtain a copy of the benchmark, please contact `fyodor@nas.nasa.gov`.

6.1 Acknowledgments

We thank MRJ Technology Solutions, Jeffrey Becker, Edward Hook, Bill Nitzberg, Doug Sakal, Parkson Wong, and Louis Zechtzer for their support, and Matthew Frigo and Steven G. Johnson for developing FFTW.

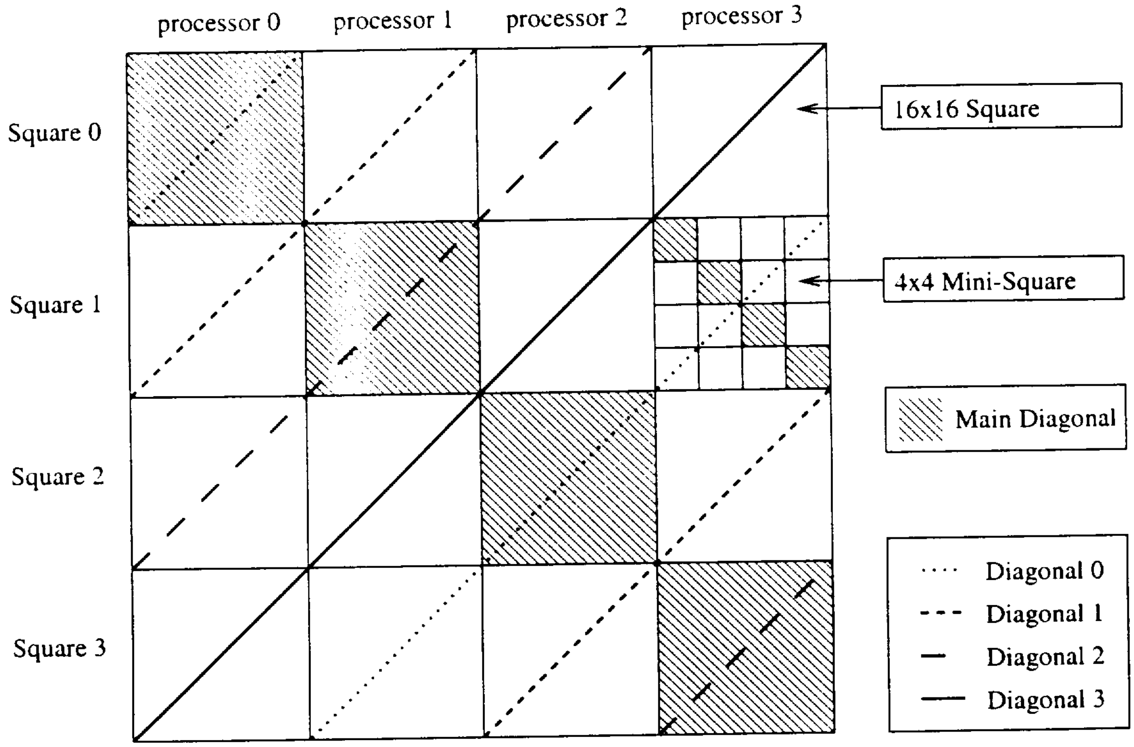


Figure 1: 64x64 Matrix on 4 processors. An illustration of the the distribution of data among the processors and organization used for transposing. Each processor stores one column of Squares. The width of a Square is $16 = 64/4$, the width of the entire matrix divided by the number of processors. A Square is composed of Mini-Squares which have width $4 = \sqrt{16}$, the square root of the Square's width. The Main Diagonal of the entire matrix, or of a Square is indicated by shading. And the Diagonals used to organize interprocessor communication are indicated by lines. For example, Diagonal 0 prompts the following actions: Square 0 on processor 0 remains stationary; Square 3 of processor 1 is exchanged with Square 1 of processor 3; and Square 2 of processor 2 remains stationary.

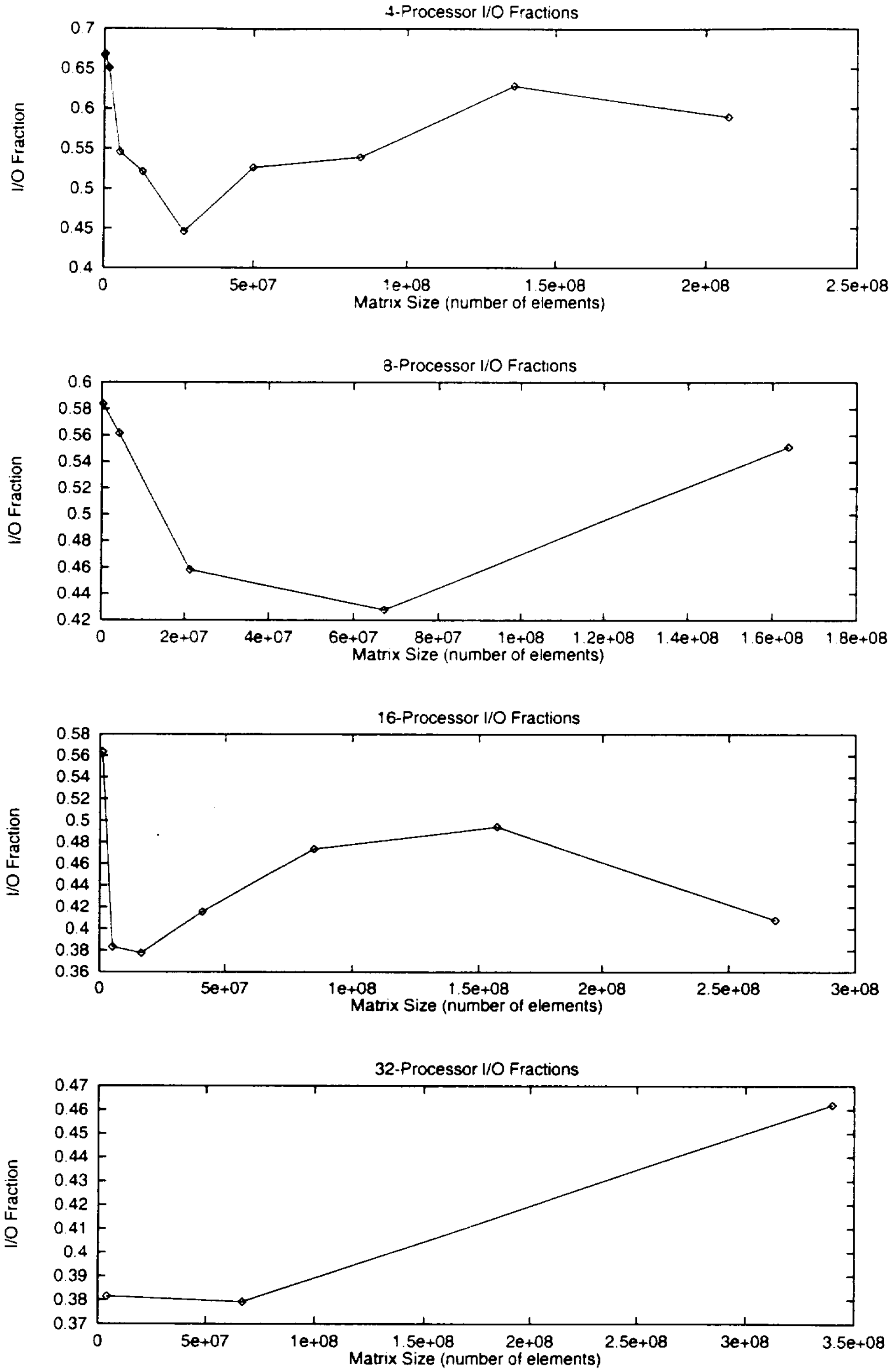


Figure 2: Estimation using I/O Fractions

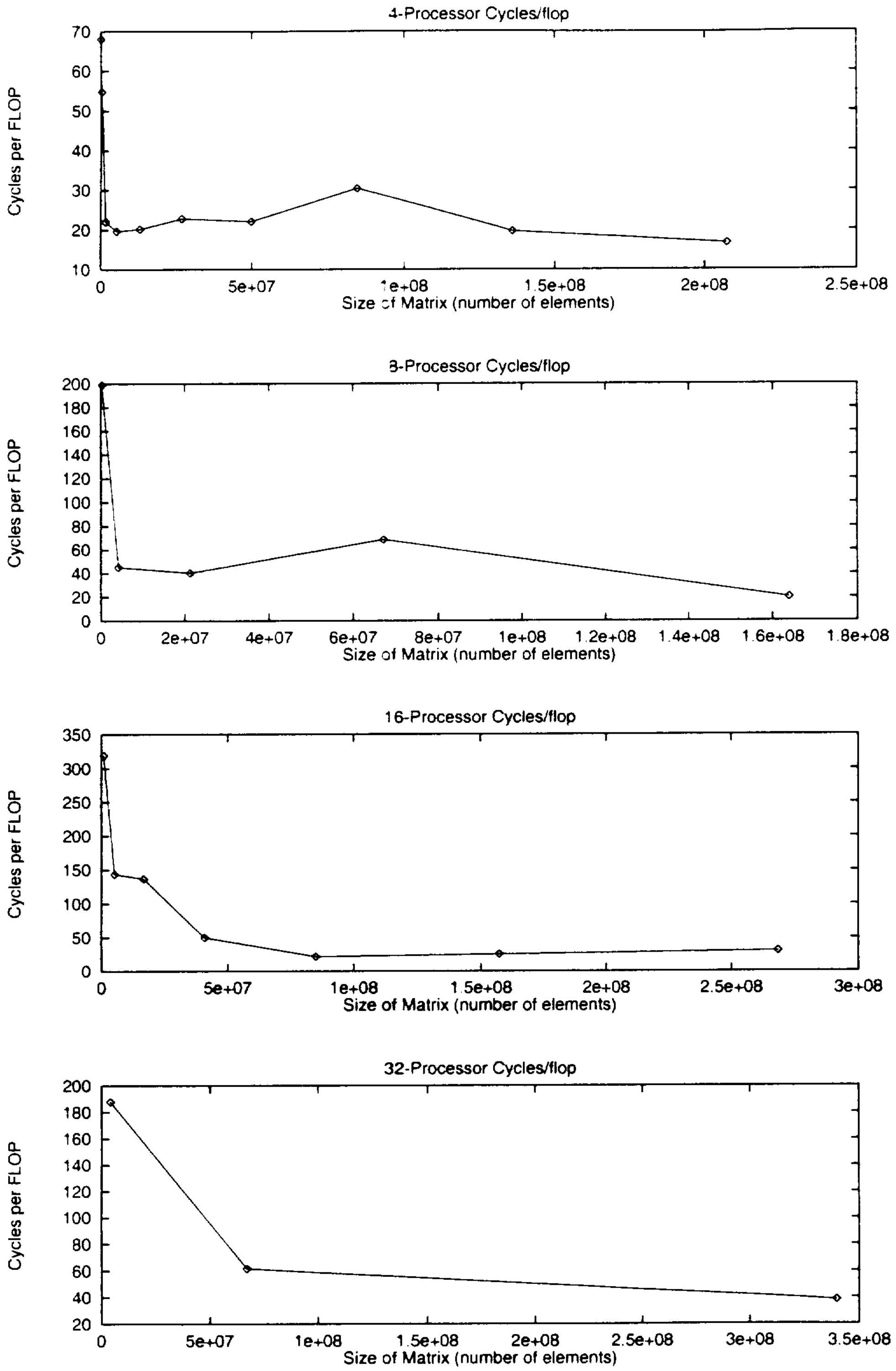


Figure 3: Estimation using Cycles/flop.

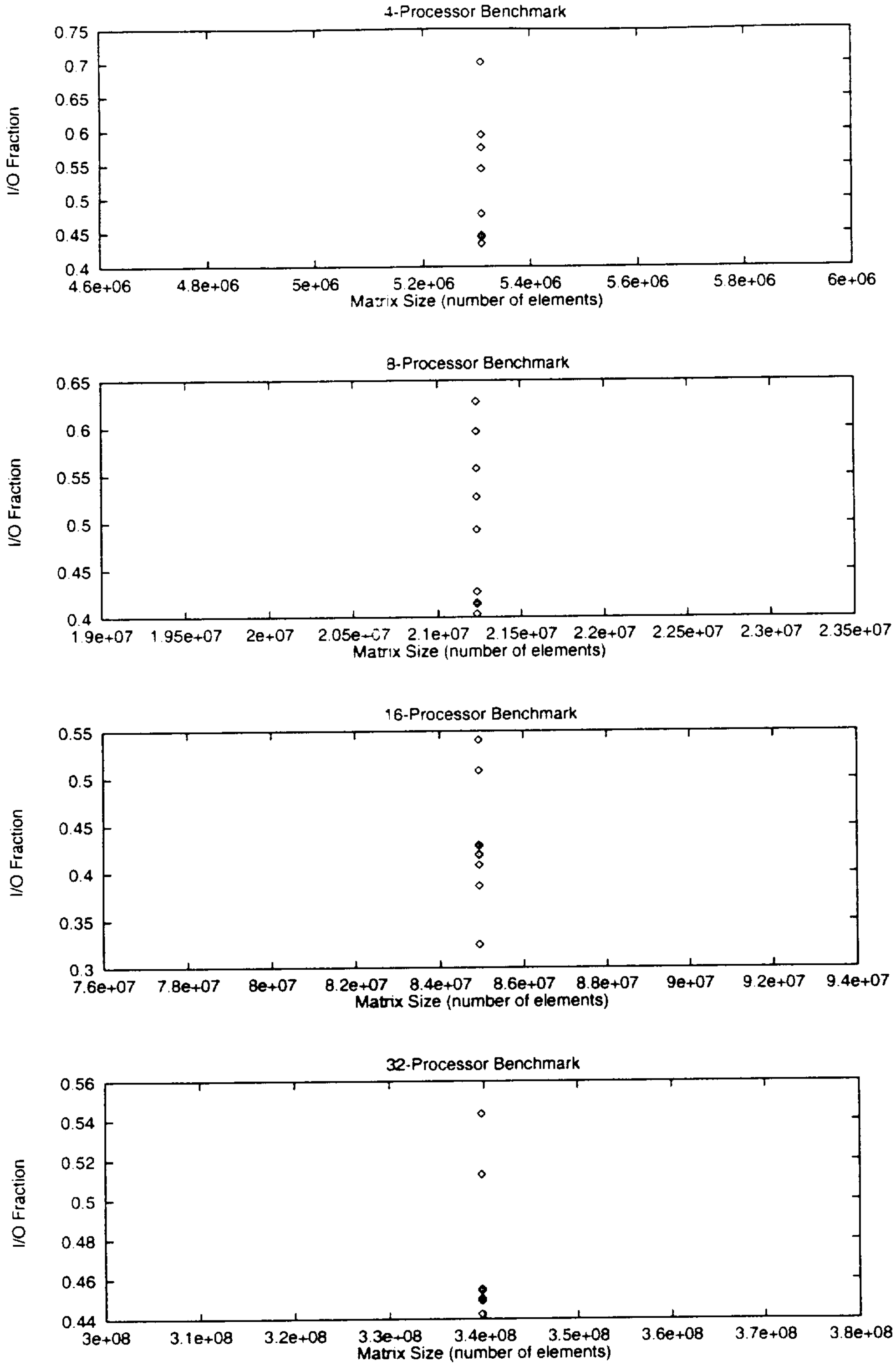
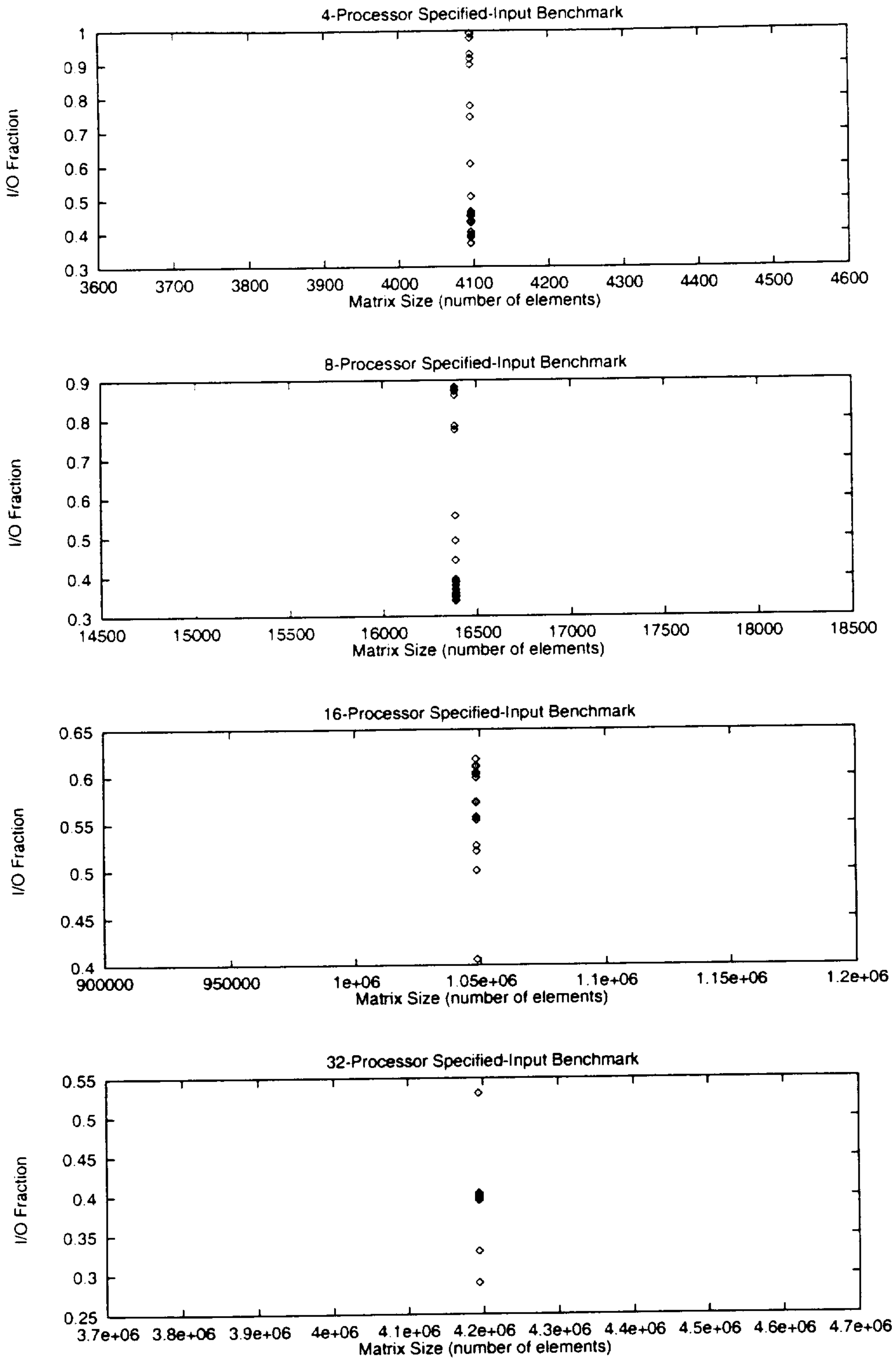


Figure 4: Benchmark Data.



4-Processor Specified-Input Benchmark

8-Processor Specified-Input Benchmark

16-Processor Specified-Input Benchmark

32-Processor Specified-Input Benchmark

Figure 5: Specified-Input Benchmark Data.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[2] Russell Carter, Bob Ciotti, Sam Fineberg, and Bill Nitzberg. NHT-1 I/O benchmarks. Technical Report RND-92-016, NAS Systems Division, NASA Ames, November 1992.

[3] P. M. Chen and D. A. Patterson. A new approach to I/O performance evaluation - self-scaling I/O benchmarks, predicted I/O performance. *ACM Transactions on Computer Systems*, 12, 4:309–339, 1994.

[4] Matteo Frigo and Steven G. Johnson. `http://www.fftw.org`.

[5] `http://www-unix.mcs.anl.gov/mpi/index.html`.

[6] `http://pbs.mrj.com/`.